5

In-GPU GNN-based Intrusion Detection System

Ahmed Salah Tawfik Ibrahim[1], Emilio Paolini[2], Filippo Cugini[1], and Francesco Paolucci[1]

[1]CNIT, Italy
[2]Scuola Superiore Sant'Anna, Italy

Abstract

Graph Neural Networks (GNNs) have proven effective for intrusion detection by modeling the relational structure of network traffic. However, the high cost of graph construction often dominates the overall prediction time, limiting real-time applicability. In this work, we propose a GPU-accelerated framework that speeds up not only GNN inference but also the graph construction phase. Unlike traditional approaches that build the adjacency matrix from scratch for each input graph, our method introduces a precomputed default adjacency matrix, generated in parallel on the GPU, which is then selectively modified for each graph instance. Node features are also computed using GPU threads, enabling end-to-end graph generation and inference entirely within GPU memory. Experimental results on the 5G-NIDD dataset show a consistent speedup of approximately 1.22× over CPU execution, while preserving detection accuracy. This work demonstrates the feasibility of deploying GNN-based intrusion detection systems in time-sensitive environments by optimizing both algorithmic design and hardware utilization.

Keywords: Graph Neural Networks (GNN), GPU Acceleration, Intrusion Detection, Cybersecurity, CUDA.

5.1 Introduction and Background

The interest in graph neural networks (GNNs) has been growing in recent years. This is thanks to their outstanding ability to model and learn from non-Euclidean data [1] like graphs, allowing them to incorporate relational information into the learning [2]. In fact, GNNs have proven useful in intrusion detection in computer networks as highlighted in [3].

Nonetheless, real-time deployment is challenged by the prediction time $T_{Prediction}$, defined as the summation of the graph construction time T_{Graph} and the inference time $T_{Inference}$, which is almost entirely dependent on T_{Graph} because $T_{Inference}$ is negligible as shown in Figure 5.1.

This paper proposes utilizing the GPU to accelerate the graph construction, reducing $T_{Prediction}$. Following the Single Instruction Multiple Data (SIMD) paradigm, the GPU executes the same instruction on multiple data items in parallel using threads [4].

As key enablers to deep learning (DL), GPUs are particularly suitable for the acceleration of matrix operations, which constitute the backbone of neural network training and inference [5]. Since matrices are used to model graphs [6], GPUs can be used to accelerate their computations.

A graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E. To model this graph, two matrices are required: the feature matrix X and the adjacency matrix A [7]. Constructing X involves organizing the features of the nodes in V such that each row of X contains the features of one node, resulting in a matrix of size $|V| \times n$ where n is the number of features. The matrix A, instead, is a binary square matrix of size $|V|$ x $|V|$ with entries reflecting the existence of edges between nodes.

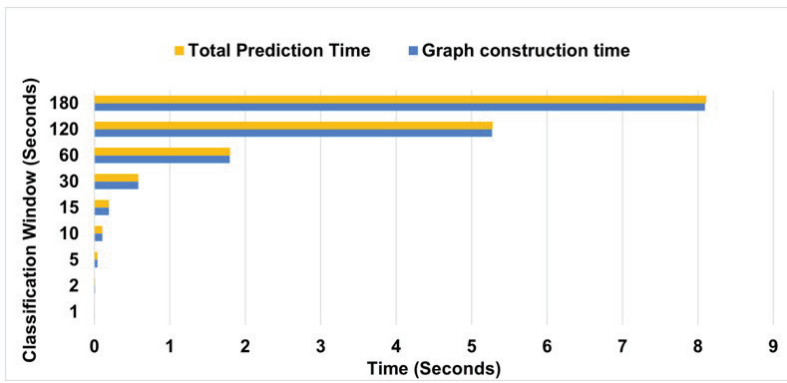

Figure 5.1 Total Prediction Time vs Graph Construction Time.

Several GPU Graph libraries exist; however, they are mostly general purpose and they do not provide the flexibility needed to construct a graph from incoming packets. For instance, systems such as cuSTINGER [8] and Hornet [9] support inserting and deleting edges while keeping the vertex set unchanged. cuSTINGER offers a flexible, dynamic adjacency-list structure on GPUs and was one of the earliest demonstrations of high-throughput GPU updates. However, its memory layout is not well suited for fine-grained, real-time insertions. Network telemetry commonly arrives as numerous small, time-ordered events rather than large, uniform batches, and cuSTINGER's dependence on batch-oriented processing can cause substantial latency increases and update stalls—problems that are especially detrimental for intrusion detection or online flow tracking.

GNNs learn from graphs by performing two operations: (i) message passing and (ii) embedding update [10]. That way, nodes aggregate their own embeddings with those of their neighbors. The aggregation can be executed in a number of ways, including graph convolutional networks (GCNs) and graph attention networks (GATs). Usually, the final aim of graph learning is node classification, edge prediction or graph classification.

In any case, choosing how to accelerate a graph algorithm depends on the application at hand. So, one must consider the patterns and trends in the application and decide how to apply GPU acceleration accordingly.

5.2 Main Text

Recalling some of the concepts in [3] is needed before outlining the proposed methodology. In [3], Raw packets are transformed into a traffic graph $G = (V, E)$ whose nodes are identified by a flow id. The flow id is defined by the source IP, the source port, the destination IP, the destination port and the protocol. Each node contains a set of packets of the node's flow id.

The construction of the graph is completed by connecting the nodes via edges. Two nodes v_1 and v_2 have an edge if:

- **R1:** the flow ids of the two nodes are different but $Src(v_1) = Dst(v_2)$ or $Src(v_2) = Dst(v_1)$.

Hence, constructing the graph can be summarized in 4 steps: (i) collecting packets; (ii) extracting features; (iii) filling the nodes; and (iv) constructing the adjacency matrix.

The initial two steps are straight-forward, so the last two steps are suitable candidates for the GPU acceleration. Adding a packet to a node involves

extracting the packet's flow id. Upon the existence of a node with the same flow id, the packet gets inserted into that node where its features are aggregated with the existing ones. Otherwise, a new node is created where the packet can be inserted. Neglecting the complexity of the aggregation, the complexity of the node creation process is linear in the number of packets ($O(|P|)$).

To construct the adjacency matrix, nodes are ordered chronologically. Then, the edge condition is checked by comparing each node with all the subsequent ones. If it holds, then the two nodes get connected by an undirected edge. This means setting the corresponding entry in the adjacency matrix to 1. Clearly, this algorithm has a quadratic complexity in the number of nodes ($O(|V|^2)$).

Upon successfully terminating these steps, the constructed matrices are passed to the GPU where the GNN inference gets performed. This transfer has an overhead that can be eliminated by constructing the graph directly on the GPU. That way, the GPU parallelization capabilities can be further utilized in speeding up the entire process.

However, achieving this paradigm shift requires handling two main practical challenges. First of all, memory pre-allocation is required in GPU programming, meaning that the number of nodes must be decided in advance. Furthermore, the adjacency matrix construction requires a unique mapping between actual IP addresses and adjacency matrix indices.

Fortunately, a classification window of 75 packets is sufficient to make accurate predictions according to [3]. In that case, the session may have up to 150 unique addresses. With this, the default adjacency matrix, defined to be the matrix corresponding to the case where there are packets between all pairs of addresses in the session, can be constructed. The size of this matrix will be $150^2 \times 150^2$ This solves the first challenge related to the pre-allocated memory.

The default matrix is invariant for all sessions of size N, so it can be built only once in the beginning. The procedure that highlights building the default matrix using the GPU is shown in Alg. 1. Each thread considers one node and fills its corresponding row and column in the matrix according to the edge condition (R1).

The second challenge requires finding a bijective function whose bijectivity holds only within the session. This means that the output of the mapping must correspond to one and only one address in the session and, in turn, each address must correspond to one and only one output. So, this bijective function can be defined as $f : x \rightarrow N$ *where* $x \in \{Src(v), Dst(v)\ \forall v \in V\}$. In

Algorithm 1 Thread to Initialize Default Adjacency Matrix

Input : Adjacency Matrix Adj, Maximum Number of Node Configurations N
Output: Initialized Adjacency Matrix Adj

```
row ← ThreadIdx.x
col ← ThreadIdx.y
if row ≠ col and row < n and col < n then
    NodeIndex ← row * N + col
    Adj[NodeIndex, NodeIndex] ← 1
    for i ← 0 to N do
        if col ≠ i then
            Adj[NodeIndex, col * N + i] ← 1
        if row ≠ i then
            Adj[NodeIndex, i * N + row] ← 1
```

the case where N = 150 addresses, the output of f should be a natural number in the set $\{0, 1, 2, ..., 149\}$.

While hashing is the preferred implementation for this mapping, it is very challenging to come up with a hash function that has the desired bijective property due to potential conflicts. Since the address space is much larger than the integers of the mapping, finding such a function is almost impossible. It is, however, possible to use an in-GPU dictionary data structure, but to the best of our knowledge, no such implementation exists in python. Therefore, a GPU-accelerated k-means clustering algorithm with k equal to the number of addresses is used. This guarantees that each address during the session will be mapped to one and only one integer between 0 and k-1, ensuring the bijective property within the session.

The constructed default adjacency matrix may be later modified by each graph, reflecting the actual flows.

To measure the performance brought by the GPU acceleration, a machine with an NVIDIA Tesla T4 GPU[1] is used to execute a number of tests.

A. Graph Initialization Time vs Window Size

The graph initialization outlined in Alg. 1 is executed to measure the time of execution. This is done by considering different classification window sizes

[1] https://www.nvidia.com/en-us/data-center/tesla-t4/

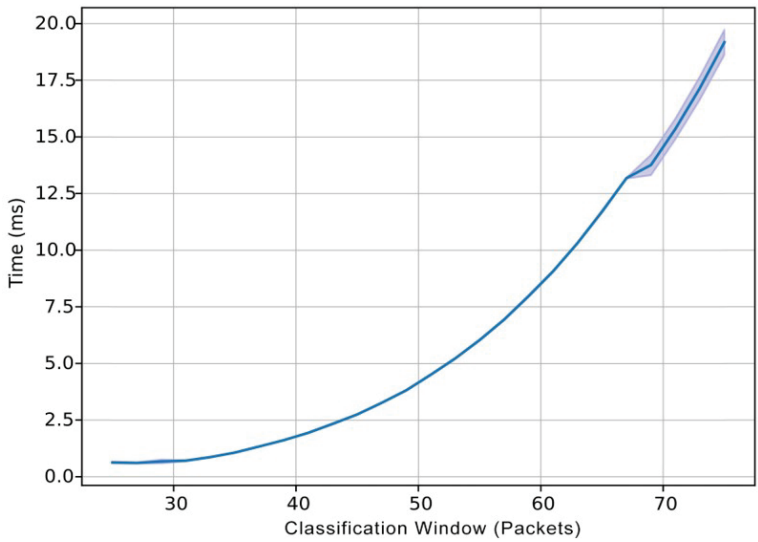

Figure 5.2 Graph Initialization Time.

Table 5.1 CPU vs GPU F1 Score

Metric	CPU	GPU
F1 Score	83.9%	83.78%

(N) and recording the execution times of each. The maximum considered size is 75 due to the GPU memory limitations. Statistical significance is ensured by executing the test 30 times to run a t-score test. Figure 5.2 reports the average of these runs along with the 95% confidence interval.

B. CPU vs GPU

In addition, it is important to measure the actual performance gain upon executing the GPU acceleration with respect to the original CPU implementation. Here, the total time to build n graphs is measured by considering different numbers of constructed graphs and recording the corresponding total construction time. The window size of the constructed graphs is set to 75 packets. They are constructed on the CPU and the GPU showing a GPU speedup of 1.22x with respect to the CPU as shown in Figure 5.3. Furthermore, it can be noted that the average F1 score, measured by averaging the F1 scores of 10,000 classification sessions for each implementation, remains almost identical as shown in Table 5.1. The slight change is attributed to the difference in floating point precision between the CPU and the GPU.

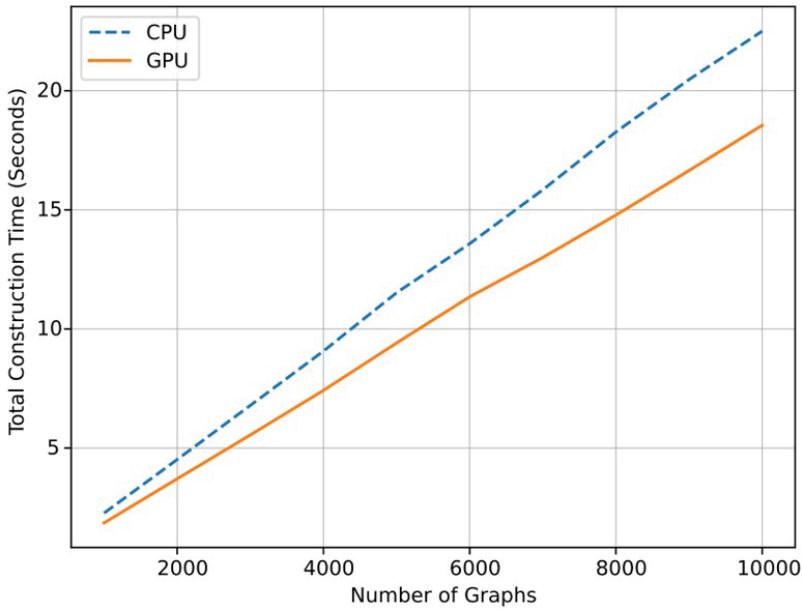

Figure 5.3 CPU vs GPU Graph Construction Time

5.3 Conclusion

In conclusion, this paper introduces GPU acceleration to the GNN intrusion detection system proposed in [3]. This acceleration speeds up the graph construction by constructing the default adjacency matrix using parallel GPU threads. At system startup, this matrix gets computed and stored to be later adjusted by each graph according to its actual edges. The improvement in performance is manifested by the fact that the accelerated system is 1.22 times faster than the original CPU-based system.

In the future, the memory requirements of the system will be considered. This is important to mitigate the exploding space requirements of the default adjacency matrix. A possible direction, therefore, is investigating sparse matrix representations to store it.

Moreover, considering GPU memory access patterns in the outlined algorithms may lead to performance gains. So, a memory-aware design of the algorithms is also another possible future direction.

Finally, to reduce the data size, using a different number representation is useful. So, instead of using 32-bit floating point numbers, 16-bit floating point representations like Bfloat16 [11] may be considered.

Acknowledgements

This work is supported by the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), including top-up funding by the Italian MUR.

References

[1] Z. Zhu, F. Li, G. Li, Z. Liu, Z. Mo, Q. Hu, X. Liang, and J. Cheng, "Mega: A memory-efficient gnn accelerator exploiting degree-aware mixed-precision quantization," in 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2024, pp. 124–138.

[2] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," IEEE transactions on neural networks and learning systems, vol. 32, no. 1, pp. 4–24, 2020.

[3] A. S. T. Ibrahim, E. Paolini, F. Cugini, and F. Paolucci, "Real-time graph neural network for malicious traffic detection," in 2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN), 2025, pp. 1–6.

[4] R. Baraglia, G. Capannini, F. M. Nardini, and F. Silvestri, "Sorting using bitonic network with cuda," CEUR Workshop Proceedings, vol. 480, 01 2009.

[5] W. Jeon, G. Ko, J. Lee, H. Lee, D. Ha, and W. W. Ro, "Chapter six - deep learning with gpus," in Hardware Accelerator Systems for Artificial Intelligence and Machine Learning, ser. Advances in Computers, S. Kim and G. C. Deka, Eds. Elsevier, 2021, vol. 122, pp. 167–215. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245820300905

[6] D. Grinberg, "An introduction to graph theory," 2025. [Online]. Available: https://arxiv.org/abs/2308.04512

[7] J. H. Tanis, C. Giannella, and A. V. Mariano, "Introduction to graph neural networks: A starting point for machine learning engineers," 2024. [Online]. Available: https://arxiv.org/abs/2412.19419

[8] O. Green and D. A. Bader, "cuSTINGER: Supporting dynamic graph algorithms for GPUs," *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2016, pp. 1-6, doi: 10.1109/HPEC.2016.7761622.

[9] F. Busato, O. Green, N. Bombieri and D. A. Bader, "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs," *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2018, pp. 1-7, doi:10.1109/HPEC.2018.8547541.

[10] W. L. Hamilton, "Graph representation learning," Synthesis Lectures on Artificial Intelligence and Machine Learning, vol. 14, no. 3, pp. 1–159.

[11] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in 2019 IEEE 26th Symposium osn Computer Arithmetic (ARITH), 2019, pp. 88–91.